Mobile Apps Final Project Report

Alex Anderson, Eric Dattore, Steve Hamrick

Overview

The Twovie Times app's goal was to grab movie times from a selected movie theater and display possible pairings of two distinct movies to the user. The inspiration of the app came from the scenario where a family is split into two groups of people where each group wants to watch a different movie. Traditionally, they would have to do the hunting for suitable movies themselves because they would like to go to the movies at the same time and get out around the same time, within a reasonable margin.

We set out to make this easier by making a mobile app that can do this for the user. To do this, we first had to find an API that would serve us the movie times we wanted. From there, we had to parse the API into a usable format and then write an algorithm to attempt to pair movies based on user input.

We did this very successfully and produced a working app that would pair movies and help the user visualize the movie time gaps on either end (start or end times). This was an interesting project because, as far as we know, nothing like this currently exists. While the project was constrained because it was a school project, continuing to polish the app and improving the user experience would no doubt be well-received by a larger user-base.

Network

Our app leveraged the networking abilities of Android applications in order to query the movie times API we decided to use. In order to do this, we had to specify the usage of said permission in the Android Manifest and then we had to make sure the user Okayed the request for permissions. Once that happened, we had to create an async task to query the API since network activity can't be executed on the main thread (as this would block the UI). From there, we parsed the resulting JSON, cached the results (more on this later), and then loaded the data into the model layer that then we leveraged elsewhere in the app.

Each of the individual subsection will elaborate, in greater detail, on the sub-components of the networking component in the app.

API

For our app, we used the Gracenote movie times API. Unfortunately, Fandango's API wasn't accepting new registrations, so we used the OnConnect API from Gracenote to query movie theater locations and movie times. We were only able to register for a free account (our request for a trial account was never responded to) which limited us to 50 API requests per day. This API call limit was mostly inconsequential, but it did influence some of our design decisions when it came to interacting with the API.

The API was accessed through an AsyncTask in each of the controllers it was used in. This decision is obviously enforced by Android since we can't execute network tasks in the main/UI thread. We used the broadest API queries possible to fetch the greatest amount of data possible. This allowed us to minimize the number of API requests we made to keep under the API request limit that was imposed on us.

We were able to query the API for theater locations using a lat/lon pair or a zip code. This fact would later influence our decision in the location usage. This also allowed us to get all the theater locations in a given geographic region without too much issue.

Caching

Because of the API query limit, we ended up using caching to ensure we were able to continue testing the app even if we exhausted the API limit for a given day and so that the user experience was improved. We would cache the last location along with the theater locations and movie times for those locations. This would enable us to continue testing in a fixed location despite exhausting the API limit. This would also improve the user experience and lessen the latency of fetching theater locations and movie times when the user launched the app in the last-used location. Using the location services and the GPS, caching obviously wouldn't take effect, but if the user was using the zip code fallback, they would take full advantage of caching and see the lower latency and faster response time of the app.

Location

We used location services to fetch the user's location and display the theaters in a fixed area. This area was user-customizable and the fallback location was customizable as well. We leveraged either a fixed zip code or a variable GPS location reported on by Google's Play Services.

GPS

The GPS was used to get exact location of the user. This was then referred to the API in the form of a latitude and longitude pair. This allowed us to get the exact location of the user and give them the best list of theaters to view movies in. This improved the user experience because it allowed the user to get their location without them inputting an address or zip code.

Once we got their location, we could also send to the API a radius limit to narrow the search down. To do this, we used a default value and allowed the user to change this value in the settings view. This allowed the user to, not only get their exact location, but specify how close they wanted the theater to be to their current location.

Fallback

As a fallback, in case they rejected location access or we couldn't get their location, we allowed them to enter a fallback zip code in the settings view. This allowed the user to retain their privacy or allow the app to work when location services were unavailable. Since the app was developed as a school project, the default zip code for the app is set to 80401, the zip code of Golden. The user has full control over changing this setting in the settings view.

Movie Pairing

The core of the application was in the movie pairing. Once we had the user's location and we pulled the list of movies at the current theater, we could allow the user to pick the movies or genres they wanted to pair on. They could also mix the two together to view more possibilities. From there, we would send a request to pair the movie titles or genres to the API Handler which then return a listing of Movies that the user could go to. This formed the core functionality of our app. It worked pretty well too. While the algorithm was fairly rudimentary, it got the job done fairly well. More work into the algorithm could have resulted in more interesting or more efficient movie pairings.

Graphics

We used Android's 2D graphics to drawn the overlap. When you look at movie pairings, you'll see lines drawn at different lengths. This is to help the user visualize the difference in start, end, and run times of the movies selected to be paired together. We ended up abstracting away the 2D graphics to an internal API that makes it simple to draw the run times of the movies. This allowed us to simply use the model layer and call a function to automatically handle the drawing. This made things much simpler and allowed us to concentrate on maintaining a clean model layer.

Challenges

The biggest challenge for this assignment was dealing with Android. By far, the API is one of the most confusing we've worked with and one of the easiest APIs to write really bad code in. The separate of model, view, and controller is very loose and the only separation enforced is between the XML layout (or view) and the Activity (or controller). The model layer could exist entirely in the controller layout if it so wished to. Also, fragments proved to be confusing and difficult to work with. Unfortunately, fragments are Android's solution to variable sized devices and the fragmentation doesn't help the ecosystem at all. In fact, it makes the development more complicated.

In the end, most of our challenges revolved around the use of Android and not from the app's actual functionality. Android, while an interesting platform, doesn't have the best developer experience and is quite annoying to work with. Partly because it's not opinionated enough and partly because of the nature of the devices your code will inevitably run on. It's a decent platform, but they really need to work harder on the developer experience.